**1. DOM 简介**

**1.1 什么是 DOM**

文档对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展标记语言（HTML 或者XML）的标准编程接口。

W3C 已经定义了一系列的 DOM 接口，通过这些 DOM 接口可以改变网页的内容、结构和样式。

l 文档：一个页面就是一个文档，DOM 中使用 document 表示

l 元素：页面中的所有标签都是元素，DOM 中使用 element 表示

l 节点：网页中的所有内容都是节点（标签、属性、文本、注释等），DOM 中使用 node 表示

**DOM 把以上内容都看做是对象**

**2.1 如何获取页面元素**

**2.2 根据 ID 获取**

document.getElementById('id');

使用 console.dir() 可以打印我们获取的元素对象，更好的查看对象里面的属性和方法。

**2.3 根据标签名获取**

使用 getElementsByTagName() 方法可以返回带有指定标签名的对象的集合。

document.getElementsByTagName('标签名');

注意：

1. 因为得到的是一个对象的集合，所以我们想要操作里面的元素就需要遍历。

2. 得到元素对象是动态的

3. 如果获取不到元素,则返回为空的伪数组(因为获取不到对象)

还可以获取某个元素(父元素)内部所有指定标签名的子元素.

element.getElementsByTagName('标签名');

注意：父元素必须是单个对象(必须指明是哪一个元素对象). 获取的时候不包括父元素自己。

var ol = document.getElementsByTagName('ol');

        console.log(ol[0].getElementsByTagName('li'));

        console.log(ol[0]);

        console.log(typeof ol[0]);

**2.4 通过 HTML5 新增的方法获取**

1. document.**getElementsByClassName(‘类名’)**；// 根据类名返回元素对象集合

2. document.**querySelector('选择器')**; // 根据指定选择器返回第一个元素对象

3. document.**querySelectorAll('选择器')**; // 根据指定选择器返回

**注意：**

querySelector 和 querySelectorAll里面的选择器需要加符号,比如:document.querySelector('#nav');

**2.5 获取特殊元素（body，html）**

**获取body元素**

1. doucumnet.body // 返回body元素对象

**获取html元素**

1. document.documentElement // 返回html元素对象

**4. 操作元素**

JavaScript 的 DOM 操作可以改变网页内容、结构和样式，我们可以利用 DOM 操作元素来改变元素里面的内容 、属性等。注意以下都是属性

**4.1 改变元素内容**

element.innerText

从起始位置到终止位置的内容, 但它去除 html 标签， 同时空格和换行也会去掉

div.innerText = '<strong>今天是：</strong> 2019';

element.innerHTML

起始位置到终止位置的全部内容，包括 html 标签，同时保留空格和换行

var box = document.querySelector('div')

box.innerHTML = '1123'

**4.2 常用元素的属性操作**

1. innerText、innerHTML 改变元素内容

2. src、href

3. id、alt、title

**4.3 表单元素的属性操作**

利用 DOM 可以操作如下表单元素的属性：

type、value、checked、selected、disabled

**4.4 样式属性操作**

我们可以通过 JS 修改元素的大小、颜色、位置等样式。

1. element.style 行内样式操作

2. element.className 类名样式操作

**注意**：

1.JS 里面的样式采取驼峰命名法 比如 fontSize、 backgroundColor

2.JS 修改 style 样式操作，产生的是行内样式，CSS 权重比较高

**注意**：

1. 如果样式修改较多，可以采取操作类名方式更改元素样式。

2. class因为是个保留字，因此使用className来操作元素类名属性

3. className 会直接更改元素的类名，会覆盖原先的类名。

<div class="first">文本</div>

        var test = document.querySelector('div');

        test.onclick = function() {

            this.className = 'first change';

        }

、

**2.1 classList 属性**

classList属性是HTML5新增的一个属性，返回元素的类名。但是ie10以上版本支持。

该属性用于在元素中添加，移除及切换 CSS 类。有以下方法

**添加类：**

element.classList.add（’类名’）；

focus.classList.add(‘current’);

**移除类：**

element.classList.remove（’类名’）;

focus.classList.remove(‘current’);

**切换类：**

element.classList.toggle（’类名’）；

focus.classList.toggle(‘current’);

注意以上方法里面，所有类名都不带点

。

 // 3. 小圆点跟随变化

        // 把ol里面li带有current类名的选出来去掉类名 remove

        ol.querySelector('.current').classList.remove('current');

        // 让当前索引号 的小li 加上 current   add

        ol.children[index].classList.add('current');

、

**4.6 自定义属性的操作**

**1. 获取属性值**

l element.属性 获取属性值。

l element.getAttribute('属性');

**区别：**

l element.属性 获取内置属性值（元素本身自带的属性）

l element.getAttribute(‘属性’); 主要获得自定义的属性 （标准） 我们程序员自定义的属性

**2. 设置属性值**

l element.属性 = ‘值’ 设置内置属性值。

l element.setAttribute('属性', '值');

**区别：**

l element.属性 设置内置属性值

l element.setAttribute(‘属性’); 主要设置自定义的属性 （标准）

**3. 移除属性**

l element.removeAttribute('属性');

**4.7 H5自定义属性**

**自定义属性目的：是为了保存并使用数据。有些数据可以保存到页面中而不用保存到数据库中。**

自定义属性获取是通过getAttribute(‘属性’) 获取。

但是有些自定义属性很容易引起歧义，不容易判断是元素的内置属性还是自定义属性。

H5给我们新增了自定义属性：

**1. 设置H5自定义属性**

H5规定自定义属性data-开头做为属性名并且赋值。

比如 <div data-index=“1”></div>

或者使用 JS 设置

element.setAttribute(‘data-index’, 2)

**2. 获取H5自定义属性**

1. 兼容性获取 element.getAttribute(‘data-index’);

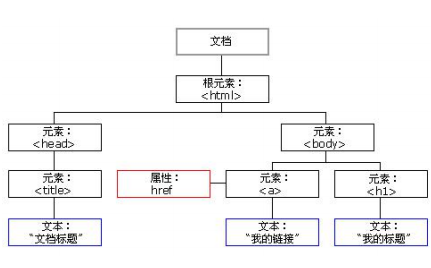
2. H5新增 element.dataset.index 或者 element.dataset[‘index’]

ie 11才开始支持

**5.2 节点概述**

网页中的所有内容都是节点（标签、属性、文本、注释等），在DOM 中，节点使用 node 来表示。

HTML DOM 树中的所有节点均可通过 JavaScript 进行访问，所有 HTML 元素（节点）均可被修改，也可以创建或删除。



**5.2 节点概述**

一般地，节点至少拥有nodeType（节点类型）、nodeName（节点名称）和nodeValue（节点值）这三个基本属性。

l 元素节点 nodeType 为 1

l 属性节点 nodeType 为 2

l 文本节点 nodeType 为 3 （文本节点包含文字、空格、换行等）

我们在实际开发中，节点操作主要操作的是**元素节点**

利用 DOM 树可以把节点划分为不同的层级关系，常见的是**父子兄层级关系**。

**1. 父级节点**

node.parentNode

l parentNode 属性可返回某节点的父节点，注意是最近的一个父节点

l 如果指定的节点没有父节点则返回 null

**2. 子节点**

1. parentNode.**childNodes（标准）**

parentNode.childNodes 返回包含指定节点的子节点的集合，该集合为即时更新的集合。

**注意：**返回值里面包含了所有的子节点，包括元素节点，文本节点等。

如果只想要获得里面的元素节点，则需要专门处理。 所以我们一般不提倡使用childNodes

var ul = document. querySelector(‘ul’);

for(var i = 0; i < ul.childNodes.length;i++) {

if (ul.childNodes[i].nodeType == 1) {

// ul.childNodes[i] 是元素节点

console.log(ul.childNodes[i]);

}

}

2. parentNode.**children（非标准）**

parentNode.children 是一个只读属性，返回所有的子元素节点。它只返回子元素节点，其余节点不返回 （**这个是我们重点掌握的**）。

虽然children 是一个非标准，但是得到了各个浏览器的支持，因此我们可以放心使用

3. parentNode.**firstChild**

firstChild 返回第一个子节点，找不到则返回null。同样，也是包含所有的节点。

4. parentNode.**lastChild**

lastChild 返回最后一个子节点，找不到则返回null。同样，也是包含所有的节点。

5. parentNode.**firstElementChild**

firstElementChild 返回第一个子元素节点，找不到则返回null。

6. parentNode.**lastElementChild**

lastElementChild 返回最后一个子元素节点，找不到则返回null。

注意：这两个方法有兼容性问题，IE9 以上才支持。

实际开发中，firstChild 和 lastChild 包含其他节点，操作不方便，而 firstElementChild 和 lastElementChild 又有兼容性问题，那么我们如何获取第一个子元素节点或最后一个子元素节点呢？

**解决方案：**

1. 如果想要第一个子元素节点，可以使用 parentNode.chilren[0]

2. 如果想要最后一个子元素节点，可以使用 parentNode.chilren[parentNode.chilren.length - 1]

**3. 兄弟节点**

1. node.**nextSibling**

nextSibling 返回当前元素的下一个兄弟元素节点，找不到则返回null。同样，也是包含所有的节点。

2. node.**previousSibling**

previousSibling 返回当前元素上一个兄弟元素节点，找不到则返回null。同样，也是包含所有的节点。

**3. 兄弟节点**

3. node.nextElementSibling

nextElementSibling 返回当前元素下一个兄弟元素节点，找不到则返回null。

4. node.previousElementSibling

previousElementSibling 返回当前元素上一个兄弟节点，找不到则返回null。

注意：这两个方法有兼容性问题， IE9 以上才支持。

**问：如何解决兼容性问题 ？**

答：自己封装一个兼容性的函数

function getNextElementSibling(element) {

var el = element;

while (el = el.nextSibling) {

if (el.nodeType === 1) {

return el;

}

}

return null;

}

**5.4 创建节点**

document.createElement('tagName')

document.createElement() 方法创建由 tagName 指定的 HTML 元素。因为这些元素原先不存在，是根据我们的需求动态生成的，所以我们也称为动态创建元素节点。

1. **node.appendChild(child)**

node.appendChild() 方法将一个节点添加到指定父节点的子节点列表末尾。类似于 CSS 里面的 after 伪元素。

2. **node.insertBefore(child, 指定元素)**

node.insertBefore() 方法将一个节点添加到父节点的指定子节点前面。类似于 CSS 里面的 before 伪元素。

**5.5 删除节点**

node.removeChild(child)

node.removeChild() 方法从 DOM 中删除一个子节点，返回删除的节点。

**5.6 复制节点(克隆节点)**

node.cloneNode()

node.cloneNode() 方法返回调用该方法的节点的一个副本。 也称为克隆节点/拷贝节点

**注意：**

1. 如果括号参数为空或者为 false ，则是浅拷贝，即只克隆复制节点本身，不克隆里面的子节点。

2. 如果括号参数为 true ，则是深度拷贝，会复制节点本身以及里面所有的子节点。

**5.8 三种动态创建元素区别**

l document.write()

l element.innerHTML

l document.createElement()

**区别**

1. document.write 是直接将内容写入页面的内容流，但是文档流执行完毕，则它会导致页面全部重绘

2. innerHTML 是将内容写入某个 DOM 节点，不会导致页面全部重绘

3. innerHTML 创建多个元素效率更高（不要拼接字符串，采取数组形式拼接），结构稍微复杂

4. createElement() 创建多个元素效率稍低一点点，但是结构更清晰

**总结**：不同浏览器下，innerHTML 效率要比 creatElement 高

function fn() {

var d1 = +new Date();

var str = '';

for (var i = 0; i < 1000; i++) {

document.body.innerHTML += '<div style="width:100px; height:2px; border:1px solid blue;"></div>';

}

var d2 = +new Date();

console.log(d2 - d1);

}

fn();

function fn() {

var d1 = +new Date();

var array = [];

for (var i = 0; i < 1000; i++) {

array.push('<div style="width:100px; height:2px; border:1px solid blue;"></div>');

}

document.body.innerHTML = array.join('');

var d2 = +new Date();

console.log(d2 - d1);

}

fn();

function fn() {

var d1 = +new Date();

for (var i = 0; i < 1000; i++) {

var div = document.createElement('div');

div.style.width = '100px';

div.style.height = '2px';

div.style.border = '1px solid red';

document.body.appendChild(div);

}

var d2 = +new Date();

console.log(d2 - d1);

}

fn();

**1. 注册事件（绑定事件）**

**1.1 注册事件概述**

给元素添加事件，称为注册事件或者绑定事件。

注册事件有两种方式：传统方式和方法监听注册方式

**传统注册方式**

l 利用 on 开头的事件 onclick

l <button onclick=“alert('hi~')”></button>

l btn.onclick = function() {}

l 特点： 注册事件的**唯一性**

l 同一个元素同一个事件只能设置一个处理函数，最后注册的处理函数将会覆盖前面注册的处理函数

**方法监听注册方式**

l w3c 标准 推荐方式

l addEventListener() 它是一个方法

l IE9 之前的 IE 不支持此方法，可使用 attachEvent() 代替

l 特点：同一个元素同一个事件可以注册多个监听器

l 按注册顺序依次执行

**1.2 addEventListener 事件监听方式**

eventTarget.addEventListener(type, listener,[ useCapture])

eventTarget.addEventListener()方法将指定的监听器注册到 eventTarget（目标对象）上，当该 对象触发指定的事件时，就会执行事件处理函数。

该方法接收三个参数：

l type：事件类型字符串，比如 click 、mouseover ，注意这里不要带 on

l listener：事件处理函数，事件发生时，会调用该监听函数

l useCapture：可选参数，是一个布尔值，默认是 false。学完 DOM 事件流后，我们再进一步学习

**1.3 attachEvent 事件监听方式**

eventTarget.attachEvent(eventNameWithOn, callback)

eventTarget.attachEvent()方法将指定的监听器注册到 eventTarget（目标对象） 上，当该对象触 发指定的事件时，指定的回调函数就会被执行。

该方法接收两个参数：

l eventNameWithOn：事件类型字符串，比如 onclick 、onmouseover ，这里要带 on

l callback： 事件处理函数，当目标触发事件时回调函数被调用

**注意：**IE8 及早期版本支持

**1.4 注册事件兼容性解决方案**

function addEventListener(element, eventName, fn) {

// 判断当前浏览器是否支持 addEventListener 方法

if (element.addEventListener) {

element.addEventListener(eventName, fn); // 第三个参数 默认是false

} else if (element.attachEvent) {

element.attachEvent('on' + eventName, fn);

} else {

// 相当于 element.onclick = fn;

element['on' + eventName] = fn;

}

兼容性处理的原则： 首先照顾大多数浏览器，再处理特殊浏览器

**2. 删除事件（解绑事件）**

**2.1 删除事件的方式**

**1. 传统注册方式**

eventTarget.onclick = null;

**2. 方法监听注册方式**

① eventTarget.removeEventListener(type, listener,[ useCapture]);

② eventTarget.detachEvent(eventNameWithOn, callback);

**2.2 删除事件兼容性解决方案**

function removeEventListener(element, eventName, fn) {

// 判断当前浏览器是否支持 removeEventListener 方法

if (element.removeEventListener) {

element.removeEventListener(eventName, fn); // 第三个参数 默认是false

} else if (element.detachEvent) {

element.detachEvent('on' + eventName, fn);

} else {

element['on' + eventName] = null;

}

**3. DOM 事件流**

事件流描述的是从页面中接收事件的顺序。

事件发生时会在元素节点之间按照特定的顺序传播，这个传播过程即 DOM 事件流。

比如我们给一个div 注册了点击事件：

DOM 事件流分为3个阶段：

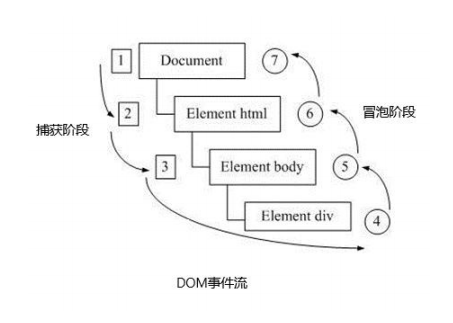
1. 捕获阶段

2. 当前目标阶段

3. 冒泡阶段

l 事件冒泡： IE 最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点的过程。

l 事件捕获： 网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收的过程。



事件发生时会在元素节点之间按照特定的顺序传播，这个传播过程即 DOM 事件流。

**注意**

1. JS 代码中只能执行捕获或者冒泡其中的一个阶段。

2. onclick 和 attachEvent 只能得到冒泡阶段。

3. addEventListener(type, listener,[ useCapture])第三个参数如果是 true，表示在事件 捕获阶段调用事件处理程序；如果是 false（不写默认就是false），表示在事件冒泡阶段调用事件 处理程序。

4. 实际开发中我们很少使用事件捕获，我们更关注事件冒泡。

5. 有些事件是没有冒泡的，比如 onblur、onfocus、onmouseenter、onmouseleave

6. 事件冒泡有时候会带来麻烦，有时候又会帮助很巧妙的做某些事件，我们后面讲解。

**4. 事件对象**

**4.1 什么是事件对象**

eventTarget.onclick = function(event) {}

eventTarget.addEventListener('click', function(event) {}）

// 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt

官方解释：event 对象代表事件的状态，比如键盘按键的状态、鼠标的位置、鼠标按钮的状态。

简单理解：事件发生后，跟事件相关的一系列信息数据的集合都放到这个对象里面，这个对象就是事件对象 event，它有很多属性和方法。

**4.2 事件对象的使用语法**

eventTarget.onclick = function(event) {

// 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt

}

eventTarget.addEventListener('click', function(event) {

// 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt

}）

这个 event 是个形参，系统帮我们设定为事件对象，不需要传递实参过去。

当我们注册事件时， event 对象就会被系统自动创建，并依次传递给事件监听器（事件处理函数）。

**4.3 事件对象的兼容性方案**

事件对象本身的获取存在兼容问题：

1. 标准浏览器中是浏览器给方法传递的参数，只需要定义形参 e 就可以获取到。

2. 在 IE6~8 中，浏览器不会给方法传递参数，如果需要的话，需要到 window.event 中获取查找。

**解决:**

e = e || window.event;

**4.4 事件对象的常见属性和方法**

e.target 和 this 的区别：

this 是事件绑定的元素， 这个函数的调用者（绑定这个事件的元素）

e.target 是事件触发的元素。

**4.4 事件对象的常见属性和方法**



**5. 阻止事件冒泡**

**5.1 阻止事件冒泡的两种方式**

事件冒泡：开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点。

事件冒泡本身的特性，会带来的坏处，也会带来的好处，需要我们灵活掌握。

**阻止事件冒泡**

l 标准写法：利用事件对象里面的 stopPropagation()方法

e.stopPropagation()

l 非标准写法：IE 6-8 利用事件对象 cancelBubble 属性

e.cancelBubble = true;

**5.2 阻止事件冒泡的兼容性解决方案**

if(e && e.stopPropagation){

e.stopPropagation();

}else{

window.event.cancelBubble = true;

}

**6. 事件委托（代理、委派）**

**事件委托**

事件委托也称为事件代理， 在 jQuery 里面称为事件委派。

**事件委托的原理**

不是每个子节点单独设置事件监听器，而是事件监听器设置在其父节点上，然后利用冒泡原理影响设置每个子节点。

以上案例：给 ul 注册点击事件，然后利用事件对象的 target 来找到当前点击的 li，因为点击 li，事件会冒泡到 ul 上， ul 有注册事件，就会触发事件监听器。

**事件委托的作用**

我们只操作了一次 DOM ，提高了程序的性能。

**7. 常用的鼠标事件**

**7.1 常用的鼠标事件**

1.禁止鼠标右键菜单

contextmenu主要控制应该何时显示上下文菜单，主要用于程序员取消默认的上下文菜单

document.addEventListener('contextmenu', function(e) {

e.preventDefault();

})

2.禁止鼠标选中（selectstart 开始选中）

document.addEventListener('selectstart', function(e) {

e.preventDefault();

})

**7.2 鼠标事件对象**

event对象代表事件的状态，跟事件相关的一系列信息的集合。现阶段我们主要是用鼠标事件对象 MouseEvent 和键盘事件对象 KeyboardEvent。



**8. 常用的键盘事件**

**8.1 常用键盘事件**

事件除了使用鼠标触发，还可以使用键盘触发。



**KeyCode 返回该键的ASCII值**

**Key 返回该键的值 是string 类型 例如返回按下a 返回的就是’a’**

**注意：**

**1. 如果使用addEventListener 不需要加 on**

2. onkeypress 和前面2个的区别是，它不识别功能键，比如左右箭头，shift 等。

3. 三个事件的执行顺序是： keydown -- keypress --- keyup

**注意：** onkeydown 和 onkeyup 不区分字母大小写，onkeypress 区分字母大小写。

在我们实际开发中，我们更多的使用keydown和keyup， 它能识别所有的键（包括功能键）

Keypress 不识别功能键，但是keyCode属性能区分大小写，返回不同的ASCII值

**1. BOM 概述**

BOM（Browser Object Model）即**浏览器对象模型**，它提供了独立于内容而与浏览器窗口进行交互的对象，其核心对象是 window。

BOM 由一系列相关的对象构成，并且每个对象都提供了很多方法与属性。

BOM 缺乏标准，JavaScript 语法的标准化组织是 ECMA，DOM 的标准化组织是 W3C，BOM 最初是Netscape 浏 览器标准的一部分。

**DOM**

l 文档对象模型

l DOM 就是把「文档」当做一个「对象」来看待

l DOM 的顶级对象是 document

l DOM 主要学习的是操作页面元素

l DOM 是 W3C 标准规范

**BOM**

l 浏览器对象模型

l 把「浏览器」当做一个「对象」来看待

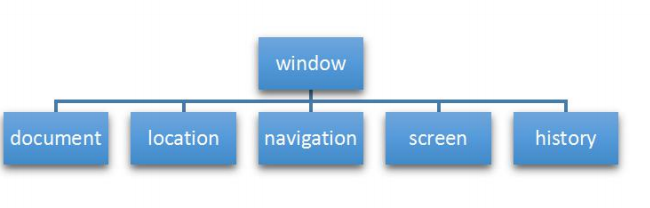
l BOM 的顶级对象是 window

l BOM 学习的是浏览器窗口交互的一些对象

l BOM 是浏览器厂商在各自浏览器上定义的，兼容性较差

**1.2 BOM 的构成**

BOM 比 DOM 更大，它包含 DOM

**1.2 BOM 的构成**

**window 对象是浏览器的顶级对象，**它具有双重角色。

1. 它是 JS 访问浏览器窗口的一个接口。

2. 它是一个全局对象。定义在全局作用域中的变量、函数都会变成 window 对象的属性和方法。 在调用的时候可以省略 window，前面学习的对话框都属于 window 对象方法，如 alert()、prompt() 等。

**注意：**window下的一个特殊属性 window.name

**2. window 对象的常见事件**

**2.1 窗口加载事件**

window.onload = function(){} 或者

window.addEventListener("load",function(){});

window.onload 是窗口 (页面）加载事件,当文档内容完全加载完成会触发该事件(包括图像、脚本文件、CSS 文件等), 就调用的处理函数。

**注意：**

1. 有了 window.onload 就可以把 JS 代码写到页面元素的上方，因为 onload 是等页面内容全部加载完毕，再去执行处理函数。

2. window.onload 传统注册事件方式 只能写一次，如果有多个，会以最后一个 window.onload 为准。

3. 如果使用 addEventListener 则没有限制

**2.1 窗口加载事件**

document.addEventListener('DOMContentLoaded',function(){})

DOMContentLoaded 事件触发时，仅当DOM加载完成，不包括样式表，图片，flash等等。

Ie9以上才支持

如果页面的图片很多的话, 从用户访问到onload触发可能需要较长的时间, 交互效果就不能实现，必然影响用户的体验，此时用 DOMContentLoaded 事件比较合适。

**2.2 调整窗口大小事件**

window.onresize = function(){}

window.addEventListener("resize",function(){});

window.onresize 是调整窗口大小加载事件, 当触发时就调用的处理函数。

**注意：**

1. 只要窗口大小发生像素变化，就会触发这个事件。

2. 我们经常利用这个事件完成响应式布局。 window.innerWidth 当前屏幕的宽度

**3.1 两种定时器**

**3.2 setTimeout() 定时器**

window.setTimeout(调用函数, [延迟的毫秒数]);

setTimeout() 方法用于设置一个定时器，该定时器在定时器到期后执行调用函数。

**注意：**

1. window 可以省略。

2. 这个调用函数可以直接写函数，或者写函数名或者采取字符串‘函数名()'三种形式。第三种不推荐

3. 延迟的毫秒数省略默认是 0，如果写，必须是毫秒。

4. 因为定时器可能有很多，所以我们经常给定时器赋值一个标识符。

**3.2 setTimeout() 定时器**

window.setTimeout(调用函数, [延迟的毫秒数]);

setTimeout() 这个调用函数我们也称为回调函数 callback

**3.3 停止 setTimeout() 定时器**

window.clearTimeout(timeoutID)

clearTimeout()方法取消了先前通过调用 setTimeout() 建立的定时器。

**注意：**

1. window 可以省略。

2. 里面的参数就是定时器的标识符 。

**3.4 setInterval() 定时器**

window.setInterval(回调函数, [间隔的毫秒数]);

setInterval() 方法重复调用一个函数，每隔这个时间，就去调用一次回调函数。

**3.5 停止 setInterval() 定时器**

window.clearInterval(intervalID);

clearInterval()方法取消了先前通过调用 setInterval()建立的定时器。

        var timer = null; // 全局变量  null是一个空对象

        begin.addEventListener('click', function() {

            timer = setInterval(function() {

                console.log('ni hao ma');

            }, 1000);

        })

        stop.addEventListener('click', function() {

            clearInterval(timer);

        })

**3.6 this**

this的指向在函数定义的时候是确定不了的，只有函数执行的时候才能确定this到底指向谁，一般情况下this 的最终指向的是那个调用它的对象

现阶段，我们先了解一下几个this指向

1. 全局作用域或者普通函数中this指向全局对象window（注意定时器里面的this指向window）

2. 方法调用中谁调用this指向谁

3.构造函数中this指向构造函数的实例

**4. JS 执行机制**

**4.1 JS 是单线程**

JavaScript 语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。这是因为 Javascript 这门脚 本语言诞生的使命所致——JavaScript 是为处理页面中用户的交互，

**4.3 同步和异步**

**同步任务**

同步任务都在主线程上执行，形成一个**执行栈。**

**异步任务**

JS 的异步是通过回调函数实现的。

一般而言，异步任务有以下三种类型:

1、普通事件，如 click、resize 等

2、资源加载，如 load、error 等

3、定时器，包括 setInterval、setTimeout 等

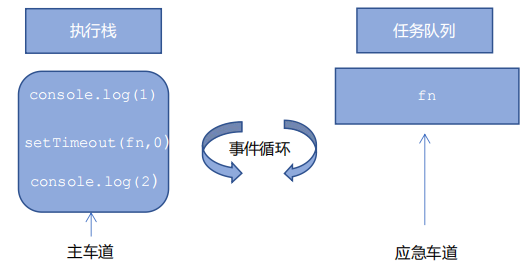
异步任务相关回调函数添加到**任务队列**中（任务队列也称为消息队列）。

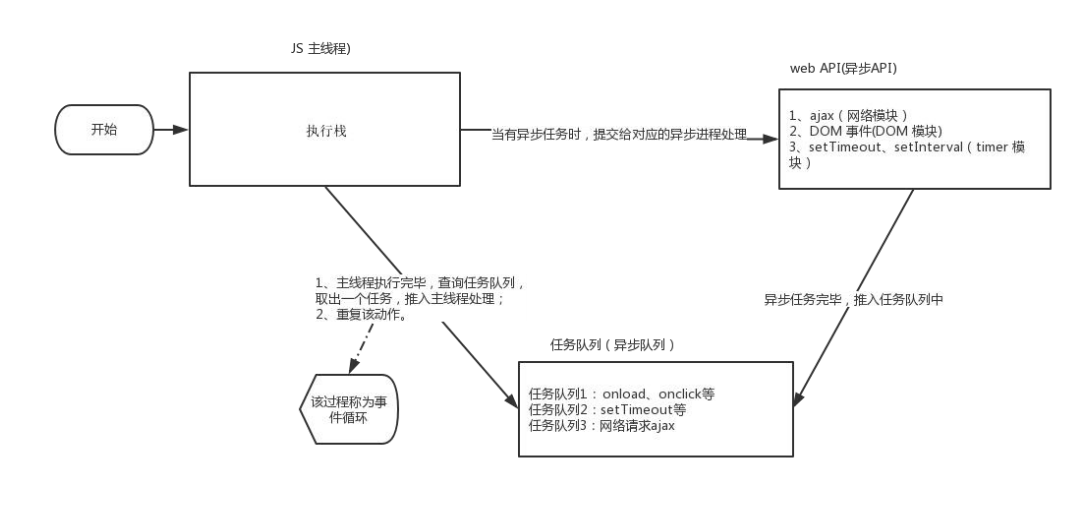
**4.4 JS 执行机制**

1. 先执行执行栈中的同步任务。

2. 异步任务（回调函数）放入任务队列中。

3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任 务结束等待状态，进入执行栈，开始执行。





由于主线程不断的重复获得任务、执行任务、再获取任务、再执行，所以这种机制被称为事件循环（ event loop 。

**5. location 对象**

**5.1 什么是 location 对象**

window 对象给我们提供了一个 location 属性用于获取或设置窗体的 URL，并且可以用于解析URL。因为这个属性返回的是一个对象，所以我们将这个属性也称为 location 对象。

**5.2 URL**

统一资源定位符 (Uniform Resource Locator, URL) 是互联网上标准资源的地址。互联网上的每个文件都有 一个唯一的 URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

URL 的一般语法格式为：

protocol://host[:port]/path/[?query]#fragment

http://www.itcast.cn/index.html?name=andy&age=18#link



**5.3 location 对象的属性**



**5.4 location 对象的方法**



**6. navigator 对象**

navigator 对象包含有关浏览器的信息，它有很多属性，我们最常用的是 userAgent，该属性可以返回由客户机发送服务器的 user-agent 头部的值。

下面前端代码可以判断用户那个终端打开页面，实现跳转

if((**navigator.userAgent**.match(/(phone|pad|pod|iPhone|iPod|ios|iPad|Android|

Mobile|BlackBerry|IEMobile|MQQBrowser|JUC|Fennec|wOSBrowser|BrowserNG|WebOS

|Symbian|Windows Phone)/i))) {

window.location.href = ""; //手机

} else {

window.location.href = ""; //电脑

}

**7. history 对象**

window 对象给我们提供了一个 history 对象，与浏览器历史记录进行交互。该对象包含用户（在浏览器窗口中） 访问过的 URL。



history 对象一般在实际开发中比较少用，但是会在一些 OA 办公系统中见到

**1. 元素偏移量 offset 系列**

**1.1 offset 概述**

offset 翻译过来就是偏移量， 我们使用 offset 系列相关属性可以动态的得到该元素的位置（偏移）、大小等。

l 获得元素距离带有定位父元素的位置

l 获得元素自身的大小（宽度高度）

l 注意： 返回的数值都不带单位

offset 系列常用属性：



**1. 元素偏移量 offset 系列**

**1.2 offset 与 style 区别**

**offset**

l offset 可以得到任意样式表中的样式值

l offset 系列获得的数值是没有单位的

l offsetWidth 包含padding+border+width

l offsetWidth 等属性是只读属性，只能获取不能赋值

l 所以，我们想要获取元素大小位置，用offset更合适

**style**

l style 只能得到行内样式表中的样式值

l style.width 获得的是带有单位的字符串

l style.width 获得不包含padding和border 的值

l style.width 是可读写属性，可以获取也可以赋值

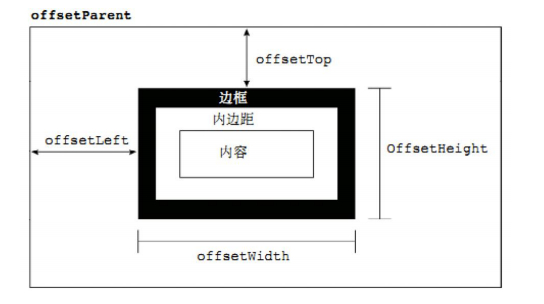
l 所以，我们想要给元素更改值，则需要用style改变

**1.1 offset 概述**

offset 翻译过来就是偏移量， 我们使用 offset 系列相关属性可以动态的得到该元素的位置（偏移）、大小等。

l 获得元素距离带有定位父元素的位置

l 获得元素自身的大小（宽度高度）



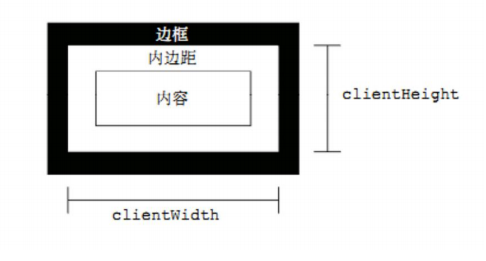
**案例：仿京东放大镜**

**2. 元素可视区 client 系列**

client 翻译过来就是客户端，我们使用 client 系列的相关属性来获取元素可视区的相关信息。通过 client 系列 的相关属性可以动态的得到该元素的边框大小、元素大小等。



client 翻译过来就是客户端，我们使用 client 系列的相关属性来获取元素可视区的相关信息。通过 client 系列 的相关属性可以动态的得到该元素的边框大小、元素大小等。

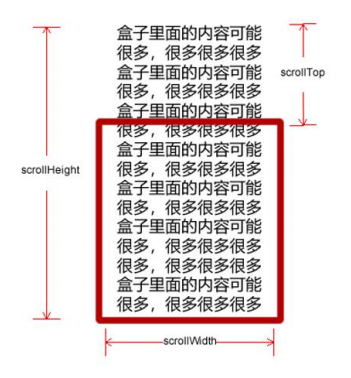


**3.1 元素 scroll 系列属性**

scroll 翻译过来就是滚动的，我们使用 scroll 系列的相关属性可以动态的得到该元素的大小、滚动距离等。



scroll 翻译过来就是滚动的，我们使用 scroll 系列的相关属性可以动态的得到该元素的大小、滚动距离等



**3.2 页面被卷去的头部**

如果浏览器的高（或宽）度不足以显示整个页面时，会自动出现滚动条。当滚动条向下滚动时，页面上面被隐藏 掉的高度，我们就称为页面被卷去的头部。滚动条在滚动时会触发 onscroll 事件。

**3. 元素滚动 scroll 系列**

**3.3 页面被卷去的头部兼容性解决方案**

需要注意的是，页面被卷去的头部，有兼容性问题，因此被卷去的头部通常有如下几种写法：

1. 声明了 DTD，使用 document.documentElement.scrollTop

2. 未声明 DTD，使用 document.body.scrollTop

3. 新方法 window. 和 window.pageXOffset，IE9 开始支持

function getScroll() {

return {

left: window.pageXOffset || document.documentElement.scrollLeft ||

document.body.scrollLeft||0,

top: window.pageYOffset || document.documentElement.scrollTop || document.body.scrollTop || 0

};

}

使用的时候 getScroll().left

**三大系列总结**



他们主要用法：

1. offset系列 经常用于获得元素位置 offsetLeft offsetTop

2. client 经常用于获取元素大小 clientWidth clientHeight

3. scroll 经常用于获取滚动距离 scrollTop scrollLeft

4. 注意页面滚动的距离通过 window.pageXOffset 获得

**mouseenter 和mouseover的区别**

**mouseenter 鼠标事件**

l 当鼠标移动到元素上时就会触发 mouseenter 事件

l 类似 mouseover，它们两者之间的差别是

l mouseover 鼠标经过自身盒子会触发，经过子盒子还会触发。 mouseenter 只会经过自身盒子触发

l 之所以这样，就是因为mouseenter不会冒泡

l 跟mouseenter搭配 鼠标离开 mouseleave 同样不会冒泡

**4. 动画函数封装**

**4.1 动画实现原理**

**核心原理：**通过定时器 setInterval() 不断移动盒子位置。

实现步骤：

1. 获得盒子当前位置

2. 让盒子在当前位置加上1个移动距离

3. 利用定时器不断重复这个操作

4. 加一个结束定时器的条件

5. 注意此元素需要添加定位，才能使用element.style.left

**4.2 动画函数简单封装**

注意函数需要传递2个参数，动画对象和移动到的距离。

   function animate (obj,target){

            clearInterval(obj.timer);

            obj.timer = setInterval(function(){

            if(obj.offsetLeft > target){

                clearInterval(obj.timer);

            }

            obj.style.left = obj.offsetLeft + 1 +'px';

        },10)

        }

        animate(div,200);

**4.3 动画函数给不同元素记录不同定时器**

如果多个元素都使用这个动画函数，每次都要var 声明定时器。我们可以给不同的元素使用不同的定时器（自己专门用自己的定时器）。

核心原理：利用 JS 是一门动态语言，可以很方便的给当前对象添加属性。

**4.4 缓动效果原理**

缓动动画就是让元素运动速度有所变化，最常见的是让速度慢慢停下来

思路：

1. 让盒子每次移动的距离慢慢变小，速度就会慢慢落下来。

2. 核心算法： (目标值 - 现在的位置 ) / 10 做为每次移动的距离 步长

3. 停止的条件是： 让当前盒子位置等于目标位置就停止定时器

4. 注意步长值需要取整

**4.5 动画函数多个目标值之间移动**

可以让动画函数从 800 移动到 500。

当我们点击按钮时候，判断步长是正值还是负值

1. 如果是正值，则步长 往大了取整

2. 如果是负值，则步长 向小了取整

**4.6 动画函数添加回调函数**

**回调函数原理**：函数可以作为一个参数。将这个函数作为参数传到另一个函数里面，当那个函数执行完之后，再执行传进去的这个函数，这个过程就叫做回调。

回调函数写的位置：定时器结束的位置。

**5.1 节流阀**

防止轮播图按钮连续点击造成播放过快。

节流阀目的：当上一个函数动画内容执行完毕，再去执行下一个函数动画，让事件无法连续触发。

核心实现思路：利用回调函数，添加一个变量来控制，锁住函数和解锁函数。

开始设置一个变量 var flag = true;

If(flag) {flag = false; do something} 关闭水龙头

利用回调函数 动画执行完毕， flag = true 打开水龙头

**节流：**减少一段时间内事件触发的频率

例如：轮播图按钮，不管按多快，规定时间内只触发一次事件

**防抖**：当触发事件时，延迟n秒再执行回调函数，如果在这n秒之内重新触发了事件，则重新计算n秒时间

**1. 本地存储**

**本地存储特性**

1、数据存储在用户浏览器中

2、设置、读取方便、甚至页面刷新不丢失数据

3、容量较大，sessionStorage约5M、localStorage约20M

4、只能存储字符串，可以将对象JSON.stringify() 编码后存储

## 2. window.sessionStorage

1、生命周期为关闭浏览器窗口

2、在同一个窗口(页面)下数据可以共享

3. 以键值对的形式存储使用

**存储数据：**

sessionStorage.setItem(key, value)

**获取数据：**

sessionStorage.getItem(key)

**删除数据：**

sessionStorage.removeItem(key)

**删除所有数据：**

sessionStorage.clear()

## 3. window.localStorage

1、声明周期永久生效，除非手动删除 否则关闭页面也会存在

2、可以多窗口（页面）共享（同一浏览器可以共享）

3. 以键值对的形式存储使用

**存储数据：**

localStorage.setItem(key, value)

**获取数据：**

localStorage.getItem(key)

**删除数据：**

localStorage.removeItem(key)

**删除所有数据：**

localStorage.clear()

set.addEventListener('click',function(){

            var val = ipt.value;

            localStorage.setItem('user',val);

        })

        get.addEventListener('click',function(){

            console.log(localStorage.getItem('user'));

        })

        remove.addEventListener('click',function(){

            localStorage.removeItem('user');

        })

        del.addEventListener('click',function(){

            localStorage.clear();

        })

#### localstorage sessionstorage cookie 区别 大小限制？(某团)

**生命周期：**

cookie：可设置失效时间，没有设置的话，默认是关闭浏览器后失效

localStorage：除非被手动清除，否则将会永久保存。

sessionStorage： 仅在当前网页会话下有效，关闭页面或浏览器后就会被清除。

**存放数据大小：**

cookie：4KB 左右

localStorage 和 sessionStorage：可以保存 5MB 的信息。

**http 请求：**

cookie：每次都会携带在 HTTP 头中，如果使用 cookie 保存过多数据会带来性能问题

localStorage 和 sessionStorage：仅在客户端（即浏览器）中保存，不参与和服务器的通信

**易用性：**

cookie：需要程序员自己封装，源生的 Cookie 接口不友好

localStorage 和 sessionStorage：源生接口可以接受，亦可再次封装来对 Object 和 Array 有更好的支持

**应用场景：**

从安全性来说，因为每次 http 请求都会携带 cookie 信息，这样无形中浪费了带宽，所以 cookie 应该尽可能少的使用，另外 cookie 还需要指定作用域，不可以跨域调用，限制比较多。但是用来识别用户登录来说，cookie 还是比 storage 更好用的。其他情况下，可以使用 storage

localStorage 和 sessionStorage 唯一的差别一个是永久保存在浏览器里面，一个是关闭网页就清除了信息。

localStorage 可以用来夸页面传递参数，sessionStorage 用来保存一些临时的数据，防止用户刷新页面之后丢失了一些参数。

**浏览器支持情况：**

localStorage 和 sessionStorage 是 html5 才应用的新特性，可能有些浏览器并不支持，IE8+

cookie 可以通过下面这段代码来判断所使用的浏览器是否支持 cookie

if(navigator.cookieEnabled) {

alert("该浏览器支持cookie"); // 浏览器支持cookie

} else {

alert("该浏览器不支持cookie"); // 浏览器不支持cookie

}

## 4. cookie

### 设置cookie

#### 设置cookie

document.cookie = 'name=张三'

#### 设置到期时间

但是我们知道 cookie 是用来保存用户的登录信息的， 我们不能让 cookie 一直在浏览器上存在，需要给 cookie 设置一个到期时间，

<script>

let data = new Date().getTime() //先将此时的时间转化为毫秒

let new\_data = new Date(data + 7\*24\*60\*60\*1000) //将过期时间设置为7天后

// toUTCString() 是将时间根据世界时转换为字符串

document.cookie = 'name=张三;' + 'expires=' + new\_data.toUTCString()

</script>

#### 多条设置

**注意**：这里补充一点，我们如果想要设置多条 cookie ， 我们必须一条一条设置，不能批量设置，例如如果要设置两条cookie的话：

<script>

let data = new Date().getTime() //先将此时的时间转化为毫秒

let new\_data = new Date(data + 7\*24\*60\*60\*1000) //将过期时间设置为7天后

//设置第一条 cookie

document.cookie = 'name=张三;' + 'expires=' + new\_data.toUTCString()

//设置第二条 cookie

document.cookie = 'age=19;' + 'expires=' + new\_data.toUTCString()</script>

### 二、查看cookie

console.log(document.cookie)

可以看到返回的是一串字符串， 每个cookie由分号隔开



#### 查看单独cookie

如果我们想要获取cookie中某个单独的 cookie，我们可以用正则进行匹配

<script>

// name=张三 ; age=19

let re = new RegExp("\s?" + "name" + "=([^;]+)(;|$)")

console.log(document.cookie.match(re)[1])

</script>

### 三、删除cookie

设置cookie是设置了一个cookie值并给它一个过期时间，这样 cookie才不会因为关闭浏览器而立马失效。所以删除 cookie 我们只需要给该cookie设置一个比现在时间还早的时间让cookie过期。

<script>

let data = new Date().getTime() //先将此时的时间转化为毫秒

let old\_data = new Date(data - 24\*60\*60\*1000) //将过期时间设为24小时前

document.cookie = 'name=张三;' + 'expires=' + old\_data.toUTCString()

</script>

我们只需要给我们需要删除的 cookie 设置一个过期时间为比现在的时间还早的时间，就可以删除这个cookie

### cookie值共享问题——域

cookie的域是用来限制哪些域名能来访问cookie的, 解决同一个主域下的访问问题

domain是cookie的一个属性, 表示的是cookie所在的域，默认为请求的地址

设置domain的格式如下:

document.cookie = "username=nO0b; path=/; domain=baidu.com"

这里的域名前可以不带点, 原因下边会介绍

作用域:

规则是:存取cookie的只能在本域名下或者子域名下才能生效

**举个小例子:**

声明三个域名:

test.com：此为顶级域名

article.test.com：二级域名

passport.test.com：二级域名

访问test.com域名时，可以存取此域名的cookie，但是不能设置二级域名的cookie,

访问article.test.com域名时，可存取此域名的cookie，也可以存取顶级域名(父域名)test.com的cookie，也就说，子域名共享顶级域名(父域名)的cookie。

在访问article.test.com域名时，不能设置passport.test.com域名的cookie，就算设置了，浏览器也不会生效该设置。

如果不设置cookie呢?

先说第一个问题，如果不显式设置cookie，那么浏览器会生成一个只针对当前域名的cookie

什么叫只针对当前域名呢？

如果有一个域名就是e.f.com.cn，在该域名下设置的cookie如果没有显示domain，在回写浏览器的时候浏览器会特殊处理,如果是火狐，你会发现该cookie的信息，有一个主机项，而域项消失了，在chrome中，虽然有域这个项，但是域的前面少了一个点，所以说如果显式声明，不管域中带不带点，到chrome中，都是带点存储的，只有非显式声明域的cookie，浏览器存储才是不带点的，问题就在这，在e.f.com.cn服务主机中生成的cookie，只有e.f.com.cn的服务器能拿到，此时子域名拿不到这个cookie。

## Dom API

### Js ondrop事件

<https://blog.csdn.net/qq_37939251/article/details/90139579>

ondrop 事件在可拖动元素或选取的文本放置在目标区域时触发。

拖放是 HTML5 中非常常见的功能。 更多信息可以查看我们 HTML 教程中的 [HTML5 拖放](https://www.runoob.com/html/html5-draganddrop.html)。

**注意：** 为了让元素可拖动，需要使用 HTML5 [draggable](https://www.runoob.com/tags/att-global-draggable.html) 属性。

**提示：** 链接和图片默认是可拖动的，不需要 draggable 属性。

在拖放的过程中会触发以下事件：

**在拖动目标上触发事件** (源元素)**:**

[ondragstart](https://www.runoob.com/jsref/event-ondragstart.html) - 用户开始拖动元素时触发

[ondrag](https://www.runoob.com/jsref/event-ondrag.html) - 元素正在拖动时触发

[ondragend](https://www.runoob.com/jsref/event-ondragend.html) - 用户完成元素拖动后触发

**释放目标时触发的事件:**

[ondragenter](https://www.runoob.com/jsref/event-ondragenter.html) - 当被鼠标拖动的对象进入其容器范围内时触发此事件

[ondragover](https://www.runoob.com/jsref/event-ondragover.html) - 当某被拖动的对象在另一对象容器范围内拖动时触发此事件

[ondragleave](https://www.runoob.com/jsref/event-ondragleave.html) - 当被鼠标拖动的对象离开其容器范围内时触发此事件

ondrop - 在一个拖动过程中，释放鼠标键时触发此事件

**注意：** 在拖动元素时，每隔 350 毫秒会触发 ondragover 事件。

拖动过程中，dragover 和 drop这两个事件一定要写的，不然不会正常触发。当外面文件往网页拖的时候会触发**加载文件的默认行为**，此时需要在这2个函数中进行阻止默认事件和阻止冒泡行为的处理。

在drop事件对象中可以通过e.dataTransfer.files 拿到拖进的文件信息，但是在控制台是看不到的，打印出来为空。但是可以进入files文件属性去查看具体文件

**for** (*const* f **of** e.dataTransfer.files) {

            console.log('File(s) you dragged here: ', f.path)

        }